Introduction to Type Theory

Damiano Testa

University of Warwick

RNTA mini symposium Atelier Lean 2023

May 3rd, 2023

Introduction to Type Theory

This talk is an extended digression on Type Theory.

As is usually the case, foundations of mathematics have a marginal impact on “real-world” mathematics.

This is true also when using Lean

... most of the times!

Set Theory

Set Theory is a common choice of foundation for mathematics.

This normally comes with

- a more or less “primitive” concept of a `set`;
- the `belongs-to` relation $\in$ among sets;
- several rules for constructing new sets from old ones;
- an empty set.

Mathematics is then built on top of these foundations.

Everything is a set

We practice Set Theory by ensuring that "everything is a set":

- the natural numbers are a set,
- ordered pairs are a set,
- the real numbers are a set,
- functions are a set,
- sequences are functions (and hence are a set),
- ...

Everything is a set. Everything. EVERYTHING.

Set Theory – really?

Most mathematicians `can` explain how to encode their favourite mathematical concept using only the basic axioms of set theory.

Most mathematicians would probably `not want` to do that.

`Descending` inside all the nested sets of sets of sets until we reach the empty set is probably even `detrimental` to developing an `intuition` about modular forms, schemes, or any `advanced` mathematical concept.

We may `undo` one nesting or two, but, after that, we probably stop and think about `structured` sets.

If $f\colon A \longrightarrow B$ is a `function`, we may think of it as a rule to `convert` elements of A to elements of B.

- No undoing: a `structured` function.

Sometimes, it can be useful to think of the `graph` of f.
In Set Theory this `is` the function.

- 1 undoing: `structured` ordered pairs.

Have you ever used Kuratowski's encoding of ordered pairs to `really` understand f?

- 2 undoings: `unstructured` chaos.

Types as structured sets

- A `Type` is like a set,
- its “elements” are called `terms`,
- the `belong-to` relation is denoted by `:` (a colon).

Thus, `t : T` means that `t` is a term of a Type `T`.

A fundamental axiom is that `every term` has a `unique` Type.

Each Type come with rules, called `constructors`, to build its terms.
The constructors endow their Type with some internal `structure`.

Let’s see the definition of natural numbers in Lean.

```
inductive myℕ
  | zero : myℕ
  | succ : myℕ → myℕ
```

Click here to open the Lean web editor.

The code above defines a Type `myℕ`.

The Type `myℕ` contains an element (really, a `term`), that we call `zero`.

We also postulate the existence of a function `succ` from `myℕ` to `myℕ`.

Lean's Type Theory takes care of making `myℕ` "universal".

For instance, Lean `auto-generates` the `induction` principle.

```
inductive myℕ
  | zero : myℕ
  | succ : myℕ → myℕ
```

In Lean's Type Theory, there is an inbuilt axiom:

- *every* term has a *unique* Type.

The Type `myℕ` contains the term `zero` (really, the term is `myℕ.zero`).

Imagine that eventually we define `myℤ.zero`.

The two terms `myℕ.zero : myℕ` and `myℤ.zero : myℤ` are *different*.

We can make Lean aware of the unique homomorphism `myℕ → myℤ`.

However, we `can't` pretend that `myℕ.zero` and `myℤ.zero` are "the same", unless some `tactic` takes care of the `conversion`.

Also that in Set Theory, the usual definitions of

$$0 \in \mathbb{N} \qquad \text{and} \qquad 0 \in \mathbb{Z}$$

yield `different` elements.

Even the `containment` $\mathbb{N} \subset \mathbb{Z}$ is `false`.

`Type Theory` simply makes us more `aware` of these (usually inconsequential) inconsistencies.

Why many proof checkers use Type Theory?

Using a proof checker, ultimately means writing a computer program to verify mathematical reasoning.

In Set Theory, **many** syntactically correct statements are `garbage`.

For instance, deciding whether the relations

$$\mathbb{N} \in \pi \qquad \text{or} \qquad \mathbb{Q}_{\leq 0} \subset e \qquad \text{or} \qquad \sqrt{2}^2 = \emptyset$$

hold is “meaningful”.

In Type Theory, none of the above `Type-checks`.

$$\mathbb{N} \in \pi \qquad \text{or} \qquad \mathbb{Q}_{\leq 0} \subset e \qquad \text{or} \qquad \sqrt{2}^2 = \emptyset$$

In the background, `Lean` constantly `Type-checks` every assertion.

This means that it can alert us to the fact that we are writing "non-sense" `before` a proof-checker based on Set Theory would.

You can think of `Type-checking` as `dimensional-analysis` in physics:

[if you compute the speed of your bike to be 12Kg, you are sure that you've made a mistake!]

... and Lean will let you know.

Implementation details

Formalizing mathematics made me focus on the separation:

Platonic world	Real-world mirror
mathematical concept	realization in set theory
abstract idea	implementation detail

`Example.` Implementations of the polynomial ring $\mathbb{Z}[x]$:

- formal, linear combinations of symbols $\{x^n\}_{n\in\mathbb{N}}$;
- "meaningful", finite $\mathbb{Z}$-linear sums of the power functions $\{x^n\}_{n\in\mathbb{N}}$;
- Finitely supported functions $\mathbb{Z} \to \mathbb{N}$ with the convolution product;
- A commutative ring with unit representing the forgetful functor

$$\begin{array}{rcl} \mathbf{CommRings} & \longrightarrow & \mathbf{Sets} \\ R & \longmapsto & R \end{array}$$

Type Theory vs Set Theory

The `distinction` between Set Theory and Type Theory as foundations for mathematics is an `implementation detail`.

Most of the times, it does not matter.

Indeed, foundations are almost invisible (unless you focus on logic).

This applies to `pen-and-paper`, as well as `formalized` mathematics.

[Lean's version of Type Theory is equiconsistent with "ZFC + there exist countably many inaccessible cardinals".]